

## Clase 5 - Recursión avanzada

---

### 1. Serie de Fibonacci

La **Serie de Fibonacci** es una secuencia matemática en la que cada número es la suma de los dos anteriores, comenzando con 0 y 1. La serie comienza de la siguiente manera:

0,1,1,2,3,5,8,13,21,34,...

Formalmente, la serie se define de la siguiente manera:

$$F(n) = \begin{cases} 0 & \text{si } n = 0, \\ 1 & \text{si } n = 1, \\ F(n-1) + F(n-2) & \text{si } n > 1. \end{cases}$$

### Implementación Recursiva

Una forma simple de implementar la serie de Fibonacci es mediante una función recursiva. A continuación, se presenta un ejemplo en pseudocódigo basado en el libro "Fundamentos de Programación - 4ta Edición" de Luis Joyanes Aguilar:

```
Función Fibonacci(n: entero): entero
    Si n == 0 Entonces
        retornar 0
    FinSi
    Si n == 1 Entonces
        retornar 1
    FinSi
    retornar Fibonacci(n - 1) + Fibonacci(n - 2)
FinFunción
```

### Análisis de Complejidad

La implementación recursiva del algoritmo de Fibonacci es sencilla y fácil de entender; sin embargo, no es eficiente debido a la gran cantidad de cálculos redundantes que realiza. Cada llamada a **Fibonacci(n)** desencadena dos llamadas recursivas adicionales: **Fibonacci(n-1)** y **Fibonacci(n-2)**. Esto provoca una explosión exponencial en el número de llamadas a la función, resultando en una complejidad temporal de  $O(2^n)$ .

Este comportamiento se debe a la repetición de los mismos cálculos para los mismos valores de n. Por ejemplo, para calcular **Fibonacci(5)**, el cálculo de **Fibonacci(3)** se realiza varias veces: una vez como parte de **Fibonacci(5)** y otra como parte de **Fibonacci(4)**.

Este enfoque recursivo, aunque conceptualmente sencillo, es muy ineficiente para valores grandes de n debido al gran número de cálculos duplicados. Una forma de optimizar este cálculo es utilizando **memoización**, que discutiremos en la Clase 6, para almacenar y reutilizar

resultados de subproblemas ya resueltos, reduciendo significativamente la complejidad temporal a  $O(n)$ .

## Ejemplo de Cálculo

Para ilustrar cómo se calcula la serie de Fibonacci de forma recursiva:

1. `fibonacci(4)` se calcula como `fibonacci(3) + fibonacci(2)`
2. `fibonacci(3)` se calcula como `fibonacci(2) + fibonacci(1)`
3. `fibonacci(2)` se calcula como `fibonacci(1) + fibonacci(0)`

Cada uno de estos cálculos se expande de nuevo hasta llegar a los casos base: `Fibonacci(0) = 0` y `Fibonacci(1) = 1`. A medida que estos valores se devuelven, se suman de nuevo hasta que se completa el cálculo original.

Esta naturaleza recursiva significa que el cálculo de Fibonacci se realiza de arriba hacia abajo, recalculando frecuentemente los mismos valores. Como resultado, `Fibonacci(2)` se calcula dos veces, `Fibonacci(1)` se calcula tres veces, etc. Esto ilustra la ineficiencia del enfoque recursivo simple sin optimización.

## ordenamiento y Búsqueda Recursivos

En el contexto de algoritmos, tanto la ordenamiento como la búsqueda pueden implementarse utilizando técnicas recursivas. A continuación, exploraremos dos algoritmos fundamentales que utilizan recursión: **ordenamiento por Mezcla (Merge Sort)** y **ordenamiento Rápida (Quick Sort)**. También veremos la **Búsqueda Binaria (Binary Search)** implementada de forma recursiva.

### ordenamiento por Mezcla (Merge Sort)

La **ordenamiento por Mezcla (Merge Sort)** es un algoritmo de ordenamiento basado en el paradigma "divide y vencerás", lo que implica dividir el problema original en subproblemas más pequeños, resolver cada subproblema de manera recursiva y luego combinar las soluciones para resolver el problema original.

En el caso de Merge Sort, el array o lista se divide repetidamente en dos mitades hasta que cada subarray tiene un solo elemento. Luego, las sublistas se combinan de forma ordenada para producir listas ordenadas de mayor tamaño. Este proceso de combinación es lo que realmente da a Merge Sort su eficiencia.

El algoritmo es particularmente útil en situaciones donde se requiere estabilidad en el ordenamiento, ya que no cambia el orden relativo de los elementos iguales. La complejidad temporal de Merge Sort es  $O(n \log n)$ , en todos los casos (mejor, peor y promedio), lo que lo hace uno de los algoritmos de ordenamiento más eficientes disponibles para listas grandes. A diferencia de otros algoritmos de ordenamiento como Quick Sort, Merge Sort garantiza siempre este rendimiento debido a la forma en que divide y combina las listas.

## Definición y Funcionamiento:

De acuerdo con Cormen et al. en *Introduction to Algorithms* (2001), "el algoritmo de ordenamiento por mezcla divide repetidamente el array en dos mitades hasta que cada subarray contiene un solo elemento, y luego las combina en un array ordenado". Este enfoque asegura que la ordenamiento se haga de manera eficiente, manteniendo el rendimiento incluso en listas desordenadas de gran tamaño.

## Pseudocódigo de ordenamiento por Mezcla:

Procedimiento MergeSort(A: lista, inicio: entero, fin: entero)

Si inicio < fin Entonces

    mitad  $\leftarrow$  (inicio + fin) / 2

    MergeSort(A, inicio, mitad)

    MergeSort(A, mitad + 1, fin)

    Mezclar(A, inicio, mitad, fin)

FinSi

FinProcedimiento

Procedimiento Mezclar(A: lista, inicio: entero, mitad: entero, fin: entero)

    n1  $\leftarrow$  mitad - inicio + 1

    n2  $\leftarrow$  fin - mitad

    L  $\leftarrow$  nueva lista de tamaño n1

    R  $\leftarrow$  nueva lista de tamaño n2

    Para i desde 0 hasta n1 - 1 hacer

        L[i]  $\leftarrow$  A[inicio + i]

    FinPara

    Para j desde 0 hasta n2 - 1 hacer

        R[j]  $\leftarrow$  A[mitad + 1 + j]

    FinPara

    i  $\leftarrow$  0, j  $\leftarrow$  0, k  $\leftarrow$  inicio

    Mientras i < n1 y j < n2 hacer

        Si L[i] <= R[j] Entonces

            A[k]  $\leftarrow$  L[i]

            i  $\leftarrow$  i + 1

        Sino

            A[k]  $\leftarrow$  R[j]

            j  $\leftarrow$  j + 1

        FinSi

        k  $\leftarrow$  k + 1

    FinMientras

    Mientras i < n1 hacer

```
A[k] ← L[i]
i ← i + 1
k ← k + 1
FinMientras
```

```
Mientras j < n2 hacer
  A[k] ← R[j]
  j ← j + 1
  k ← k + 1
FinMientras
FinProcedimiento
```

## Ejemplo Práctico:


Supongamos que tenemos una lista  $A=[38,27,43,3,9,82,10]$ . Aplicando el algoritmo de Merge Sort, primero se divide la lista repetidamente hasta que cada sublista tiene un solo elemento:

- Divide en:  $[38,27,43,3]$  y  $[9,82,10]$
- Divide cada mitad:  $[38,27]$ ,  $[43,3]$ ,  $[9,82]$ ,  $[10]$
- Sigue dividiendo:  $[38]$ ,  $[27]$ ,  $[43]$ ,  $[3]$ ,  $[9]$ ,  $[82]$ ,  $[10]$

Luego se combina y ordena:

- $[27,38]$  y  $[3,43]$
- $[9,10]$  y  $[82]$
- Combina en:  $[3,27,38,43]$  y  $[9,10,82]$
- Lista final ordenada:  $[3,9,10,27,38,43,82]$

## Explicacion Merge sort (Chio)

 Merge Sort | Ordenamiento por mezcla

## ordenamiento Rápida (Quick Sort)

El **ordenamiento Rápido (Quick Sort)** es un algoritmo de ordenamiento altamente eficiente que utiliza el paradigma "divide y vencerás". A diferencia de Merge Sort, Quick Sort no requiere espacio adicional significativo para combinar sublistas, ya que realiza la ordenamiento en su lugar.

El proceso de Quick Sort involucra seleccionar un elemento "pivote" y reorganizar los elementos de la lista de modo que todos los elementos menores que el pivote se coloquen a su izquierda y todos los elementos mayores a su derecha. Este paso se llama "partición". Luego, Quick Sort se aplica recursivamente a las sublistas de elementos menores y mayores alrededor del pivote.

Aunque Quick Sort tiene una complejidad promedio de  $O(n \log n)$ , su complejidad en el peor de los casos es  $(n^2)$ , que ocurre cuando el pivote seleccionado es el más grande o el más pequeño de la lista, como en una lista ya ordenada. Para mitigar este problema, se pueden utilizar estrategias de selección de pivote más sofisticadas, como elegir un pivote aleatorio o utilizar el pivote mediano

## Definición y Funcionamiento:

Según Joyanes Aguilar en *Fundamentos de Programación - 4ta Edición* (2008), "Quick Sort es un algoritmo de ordenamiento que, si bien tiene un caso peor poco favorable, en promedio funciona muy rápido y es altamente eficiente debido a su bajo requerimiento de memoria y su capacidad para ordenar 'en el lugar' sin necesidad de listas adicionales".

## Pseudocódigo de ordenamiento Rápido:

Procedimiento QuickSort(A: lista, bajo: entero, alto: entero)

```
Si bajo < alto Entonces
  p ← Particionar(A, bajo, alto)
  QuickSort(A, bajo, p - 1)
  QuickSort(A, p + 1, alto)
FinSi
```

FinProcedimiento

Función Particionar(A: lista, bajo: entero, alto: entero): entero


```
pivote ← A[alto]
i ← bajo - 1


Para j desde bajo hasta alto - 1 hacer
  Si A[j] <= pivote Entonces
    i ← i + 1
    Intercambiar A[i] con A[j]
  FinSi
FinPara
```

```
Intercambiar A[i + 1] con A[alto]
retornar i + 1
```

FinFunción

 Python - Nivel 31 - Reto 6 - Ordenamiento rápido - Quicksort - Introducción

 Python - Nivel 31 - Reto 7 - Ordenamiento rápido - Quicksort - Versión simple

 Quick Sort | Ordenamiento Rápido | Explicado con Cartitas!

## Ejemplo Práctico:

Tomemos la misma lista  $A=[38,27,43,3,9,82,10]$

1. Seleccionamos el último elemento como pivote: 10
2. Reorganizamos la lista:  $[3,9,10,43,27,82,38]$ .
3. Ahora, aplicamos Quick Sort recursivamente a las sublistas:  $[3,9]$  y  $[43,27,82,38]$ .
4. La lista se ordena completamente a través de sucesivas particiones y reordenamientos.

## Búsqueda Binaria (Binary Search)

La **Búsqueda Binaria**, como vimos en la unidad 1, es un algoritmo eficiente para encontrar un elemento en una lista ordenada. La idea es dividir repetidamente la lista en mitades y descartar la mitad donde el elemento no puede estar. Este proceso continúa hasta que se encuentra el elemento o se reducen todas las posibilidades. La búsqueda binaria tiene una complejidad temporal de  $O(\log n)$ , ya que con cada comparación se reduce el tamaño del problema a la mitad. Este algoritmo es muy eficiente para listas grandes, pero requiere que la lista esté previamente ordenada.

### Definición y Funcionamiento:

Como describe López de la Fuente en *Algoritmos y estructuras de datos en Python* (2020), "La búsqueda binaria es un ejemplo clásico de cómo utilizar la recursión para dividir y conquistar. Se basa en comparar el elemento buscado con el elemento medio del array y reducir el problema a la mitad en cada paso".

### Pseudocódigo de Búsqueda Binaria:

Función BusquedaBinaria(A: lista, bajo: entero, alto: entero, objetivo: entero): entero

    Si bajo  $\leq$  alto Entonces

        medio  $\leftarrow$  (bajo + alto) / 2

        Si A[medio] == objetivo Entonces

            retornar medio

        Sino Si A[medio] > objetivo Entonces

            retornar BusquedaBinaria(A, bajo, medio - 1, objetivo)

        Sino

            retornar BusquedaBinaria(A, medio + 1, alto, objetivo)

        FinSi

    FinSi

    retornar -1 // No encontrado

FinFunción